

图文 083、动手实验：Metaspace区域内存溢出的时候，应该如何解决？

686 人次阅读 2019-09-26 07:00:00

详情 评论

动手实验：
Metaspace区域内存溢出的时候，应该如何解决？

狸猫技术窝专栏上新，基于**真实订单系统**的消息中间件（mq）实战，重磅推荐：



未来3个月，我的好朋友原子弹大侠将带你一起，全程实战，360度死磕MQ

(点击下方蓝字进行试听)

[从 0 开始带你成为消息中间件实战高手](#)

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情在评论区留言提问，我都会逐一答疑

(ps：评论区还精选了一些小伙伴对**专栏每日思考题的作答**，有的答案真的非常好！大家可以通过看别人的思路，启发一下自己，从而加深理解)

如何加群：购买了狸猫技术窝专栏的小伙伴都可以加入**狸猫技术交流群**。

(群里有不少**一二线互联网大厂**的助教，大家可以一起讨论交流各种技术)

具体**加群方式**请参见文末。

(**注：**以前通过其他专栏加过群的同学就不要重复加了)



狸猫技术窝

进店逛

相关频道



从 0 开始
战高手
已更新1

1、前文回顾

上一讲已经说了我们处理OOM需要的一些参数，今天我们来讲一下Metaspace区域内内存溢出

我们先分析一下GC日志，然后再让JVM自动dump出来内存快照，最后用MAT来分析一下这份内存快照，从内存快照里去找找到内存溢出的原因。

2、示例代码

首先我们先上之前的那段代码：

```
public class Demo1 {

    public static void main(String[] args) {
        long counter = 0;

        while(true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(Car.class);
            enhancer.setUseCache(false);
            enhancer.setCallback(new MethodInterceptor() {
                public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {
                    if(method.getName().equals("run")) {
                        System.out.println("启动汽车之间，先进行自动的安全检查.....");
                        return methodProxy.invokeSuper(o, objects);
                    } else {
                        return methodProxy.invokeSuper(o, objects);
                    }
                }
            });

            Car car = (Car) enhancer.create();
            car.run();

            System.out.println("目前创建了" + (++counter) + "个Car类的子类了");
        }

        static class Car {

            public void run() {
                System.out.println("汽车启动，开始行使.....");
            }

        }

        static class SafeCar extends Car {

            @Override
            public void run() {
                System.out.println("汽车启动，开始行使.....");
                super.run();
            }

        }

    }
}
```

我们还是用这段代码来说明，但是要记得需要在JVM参数中加入一些东西，因为我们想看一下GC日志和导出内存快照，如下所示：

```
-XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
-XX:MetaspaceSize=10m
-XX:MaxMetaspaceSize=10m
-XX:+PrintGCDetails
-Xloggc:gc.log
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=./
```

大家要注意，上面那个HeapDumpPath参数我给调整为当前项目的根目录下了，这样我们看的时候方便一些。

3、分析GC日志

接着我们用上述JVM参数运行这段程序，会发现项目下面多了两个文件，一个是gc.log，还有一个是java_pid910.hprof

当然不同的机器运行这个hprof文件的名字是不太一样的，因为他会用你的PID进程id作为文件名字。

接着我们先来分析一下gc.log，也就是分析一下他是怎么拼命往Metaspace区域里放入大量生成的类，然后触发Full GC，接着回收Metaspace区域，回收后还是无法放下更多的类，接着才会抛出内存溢出的异常。

然后我们再用MAT分析一下OOM的时候的内存快照，带着大家学习一下如何用MAT工具找到Metaspace内存溢出问题的原因。

先把我这里的GC日志给大家抛出来，同时我们就跟着GC日志一行一行分析，到底是怎么回事，大家紧紧跟着脚步来走。

```
0.716: [GC (Allocation Failure) 0.717: [ParNew: 139776K->2677K(157248K), 0.0038770 secs] 139776K->2677K(506816K), 0.0041376 secs] [Times: user=0.03 sys=0.01, real=0.00 secs]
```

大家看这行日志，这是第一次GC，他本身是一个Allocation Failure的问题

也就是说，他是在Eden区中分配对象时，发现Eden区内存不足了，于是就触发了一次ygc。

那么，这个对象到底是什么对象？

简单，还记得我们在代码里写的么？Enhancer本身是一个对象，他是用来生成类的，如下所示：Enhancer enhancer = new Enhancer()。

接着我们基于每次Enhancer生成的类还会生成那个类的对象，如下所示：Car car = (Car) enhancer.create()。

因此上述代码不光是动态生成类，本身他也是对应很多对象的，因此你在while(true)循环里不停的创建对象，当然会塞满Eden区了，大家看上述日志：[ParNew: 139776K->2677K(157248K), 0.0038770 secs]

这就是说，在默认的内存分配策略下，年轻代一共可用空间是150MB左右，这里还包含了一点Survivor区域的大小

然后大概都用到140MB了，也就是Eden区都塞满了，此时就触发了Allocation Failure，没Eden区的空间分配对象了，此时就触发ygc。

这个倒没什么可说的，因为之前我们都讲过了。

```
0.771: [Full GC (Metadata GC Threshold) 0.771: [CMS: 0K->2161K(349568K), 0.0721349 secs] 20290K->2161K(506816K), [Metaspace: 9201K->9201K(1058816K)], 0.0722612 secs] [Times: user=0.12 sys=0.03, real=0.08 secs]
```

接着我们来看这次GC，这就是Full GC了，而且通过“Metadata GC Threshold”清晰看到，是Metaspace区域满了，所以触发了Full GC

这个时候看下面的日志，20290K->2161K(506816K)，这个就是说堆内存（年轻代+老年代）一共是500MB左右，然后有20MB左右的内存被使用了，这个必然是年轻代用的。

然后Full GC必然会带着一Young GC，因此这次Full GC其实是执行了ygc了，所以回收了很多对象，剩下了2161KB的对象，这个大概就是JVM的一些内置对象了。

然后直接就把这些对象放入老年代，为什么呢，因为下面的日志：[CMS: 0K->2161K(349568K), 0.0721349 secs]

这里明显说了，Full GC带着CMS进行了老年代的Old GC，结果人家本来是0KB，什么都没有，然后从年轻代转移来了2161KB的对象，所以老年代变成2161KB了。

接着看日志：[Metaspace: 9201K->9201K(1058816K)]

此时Metaspace区域已经使用了差不多9MB左右的内存了，此时明显是发现离我们限制的10MB内存很接近了，所以触发了Full GC，但是对Metaspace GC后发现类全部存活了，因此还是剩余9MB左右的类在Metaspace里。

0.843: [Full GC (Last ditch collection) 0.843: [CMS: 2161K->1217K(349568K), 0.0164047 secs] 2161K->1217K(506944K), [Metaspace: 9201K->9201K(1058816K)], 0.0165055 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

接着又是这次Full GC，人家也说的很清晰了，Last ditch collection

就是说，最后一次拯救的机会了，因为之前Metaspace回收了一次但是没有类可以回收，所以新的类无法放入Metaspace了。

所以再最后试一试Full GC，能不能回收掉一些

结果如下: [Metaspace: 9201K->9201K(1058816K)], 0.0165055 secs]

Metaspace区域还是无法回收掉任何的类，几乎还是占满了我们设置的10MB左右。

0.860: [GC (CMS Initial Mark) [1 CMS-initial-mark: 1217K(349568K)] 1217K(506944K), 0.0002251 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

0.860: [CMS-concurrent-mark-start]

0.878: [CMS-concurrent-mark: 0.003/0.018 secs] [Times: user=0.05 sys=0.01, real=0.02 secs]

0.878: [CMS-concurrent-preclean-start]

Heap

par new generation total 157376K, used 6183K [0x00000005ffe00000, 0x000000060a8c0000, 0x0000000643790000)

eden space 139904K, 4% used [0x00000005ffe00000, 0x0000000600409d48, 0x00000006086a0000)

from space 17472K, 0% used [0x00000006086a0000, 0x00000006086a0000, 0x00000006097b0000)

to space 17472K, 0% used [0x00000006097b0000, 0x00000006097b0000, 0x000000060a8c0000)

concurrent mark-sweep generation total 349568K, used 1217K [0x0000000643790000, 0x0000000658cf0000, 0x00000007ffe00000)

Metaspace used 9229K, capacity 10146K, committed 10240K, reserved 1058816K

class space used 794K, capacity 841K, committed 896K, reserved 1048576K

接着就直接JVM退出了，退出的时候就打印出了当前内存的一个情况，年轻代和老年代几乎没占用，但是Metaspace的capacity是10MB，使用了9MB左右，无法再继续使用，所以触发了内存溢出。

此时就会在控制台打印出如下的一行东西：

Caused by: java.lang.OutOfMemoryError: Metaspace

at java.lang.ClassLoader.defineClass1(Native Method)

at java.lang.ClassLoader.defineClass(ClassLoader.java:763)

... 11 more

明确抛出异常，说OutOfMemoryError，原因就是Metaspace区域满了导致的。

因此假设是Metaspace内存溢出了，然后客服通知了我们，或者我们自己监控到了异常，此时直接去线上机器看一下GC日志和异常信息就可以了，通过上述分析立刻就知道了，系统是如何运行的，触发了几次GC之后引发了内存溢出。

4、分析内存快照

当我们知道是Metaspace引发的内存溢出之后，立马就可以把内存快照文件从线上机器拷回本地笔记本电脑，打开MAT工具进行分析，如下图所示：

▼ Problem Suspect 1

The classloader/component "**sun.misc.Launcher\$AppClassLoader @ 0x64379e4e8**" occupies **434,504 (40.70%)** bytes. The memory is accumulated in one instance of "**java.lang.Object[]**" loaded by "**<system class loader>**".

Keywords























java.lang.Object[]

sun.misc.Launcher\$AppClassLoader @ 0x64379e4e8

[Details »](#)

从这里可以看到实例最多的就是AppClassLoader

为啥有这么多的ClassLoader呢？一看就是CGLIB之类的东西在动态生成类的时候搞出来的，我们可以点击上图的Details进去看看。

| Class Name | Shallow Heap | Retained Heap | Percentage |
|--|--------------|---------------|------------|
|  sun.misc.Launcher\$AppClassLoader @ 0x64379e4e8 | 88 | 434,504 | 40.70% |
|  java.util.Vector @ 0x6437f04f0 | 32 | 390,704 | 36.59% |
|  java.lang.Object[1280] @ 0x6438018e0 | 5,136 | 390,672 | 36.59% |
|  class net.sf.cglib.core.EmitUtils @ 0x6437a1188 | 96 | 5,488 | 0.51% |
|  class net.sf.cglib.core.Constants @ 0x6437a1088 | 152 | 4,688 | 0.44% |
|  class net.sf.cglib.proxy.MethodInterceptorGenerator @ 0x6437f9568 | 80 | 4,080 | 0.38% |
|  class net.sf.cglib.reflect.FastClassEmitter @ 0x643808018 | 64 | 3,240 | 0.30% |
|  class net.sf.cglib.core.ReflectUtils @ 0x6437d9c58 | 48 | 3,080 | 0.29% |
|  class net.sf.cglib.core.CodeEmitter @ 0x6437a71b8 | 104 | 2,160 | 0.20% |
|  class net.sf.cglib.core.TypeUtils @ 0x6437a12a8 | 16 | 1,616 | 0.15% |
|  class net.sf.cglib.proxy.CallbackInfo @ 0x6437f9420 | 8 | 1,408 | 0.13% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_245 @ 0x6437900c0 | 64 | 1,304 | 0.12% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_246 @ 0x643790ad8 | 64 | 1,304 | 0.12% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_247 @ 0x6437914f0 | 64 | 1,304 | 0.12% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_248 @ 0x643791f08 | 64 | 1,304 | 0.12% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_249 @ 0x643792920 | 64 | 1,304 | 0.12% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_250 @ 0x643793438 | 64 | 1,304 | 0.12% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_251 @ 0x643793e50 | 64 | 1,304 | 0.12% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_252 @ 0x643794868 | 64 | 1,304 | 0.12% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_253 @ 0x643795280 | 64 | 1,304 | 0.12% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_254 @ 0x643795c98 | 64 | 1,304 | 0.12% |
|  class com.limao.demo.jvm.Demo1\$Car\$\$EnhancerByCGLIB\$\$7e5aa3a5_255 @ 0x6437966b0 | 64 | 1,304 | 0.12% |

为什么这里有一大堆咱们自己的Demo1中动态生成出来的Car\$\$EnhancerByCGLIB的类呢？

看到这里就真相大白了，上图已经清晰告诉我们，是我们自己的哪个类里搞出来了一大堆的动态生成的类，所以填满了Metaspace区域。

所以此时直接去代码里排查动态生成类的代码即可。

解决这个问题的办法也很简单，直接对Enhancer做一个缓存，只有一个，不要无限制的去生成类就可以了。

5、本文总结

今天这篇文章，带着大家全程基于示例代码从GC日志到内存快照进行了一通分析

从GC日志我们知道系统是如何在多次GC之后无奈内存溢出的

从内存快照我们就知道到底是什么东西占据了太多的内存，然后代码里找到原因解决即可。

希望大家跟着文章，也在自己本地实战一把，这样才能真正消化这些内容，转化为自己的东西。

End

专栏版权归公众号**狸猫技术窝**所有

未经许可不得传播，如有侵权将追究法律责任

如何加群？

添加微信号：Lvgu0715_（微信名：绿小九），狸猫技术窝管理员

发送 Jvm专栏的购买截图

由于是人工操作，发送截图后请耐心等待被拉群